Scalable Defect Detection

Manuvir Das, Zhe Yang, Daniel Wang
Center for Software Excellence
Microsoft Corporation

Part III
Lightweight Specifications for Win32 APIs

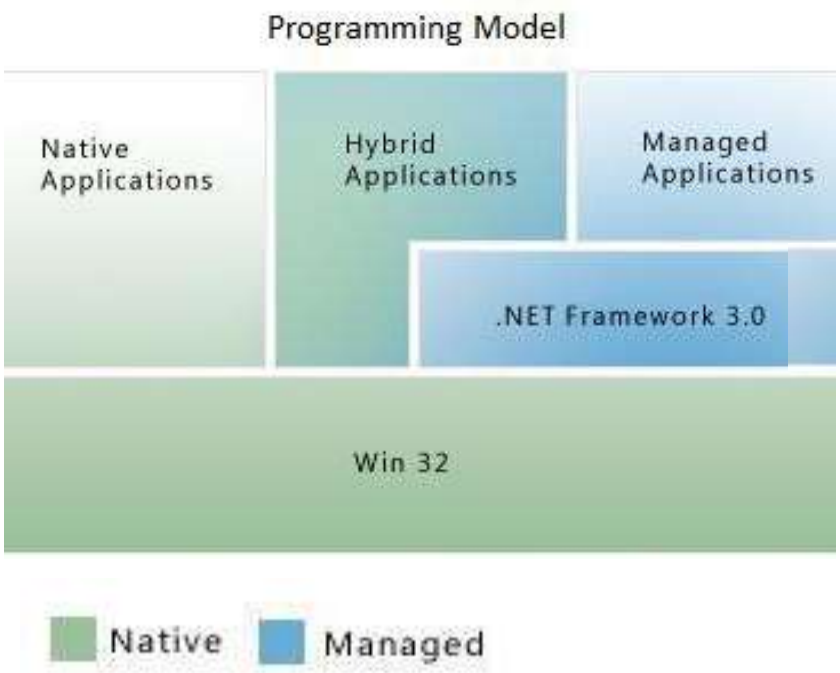
Center for Software Excellence

daniwang@microsoft.com

The Win32 API

Win32 API is the layer on which all modern Windows applications are built

.NET is built on top, and contains many managed classes that wrap Win32 functionality



Business Goals

I. Significantly reduce the number of exploitable buffer overruns in Windows Vista

II. Change development process so products after Vista are more secure

Standard Annotation Language

- Created in summer June 2002 joint effort with product groups and CSE
- Specifies *programmer intent* which leads to:
  - Better coverage (reduce false negatives)
  - Reduced noise (reduce false positives)
  - Ecosystem of tools
  - High impact results

Measured Outcomes

	Mutable String Arguments	Functions
Total	1096	20,928
Annotated	1031	6,918

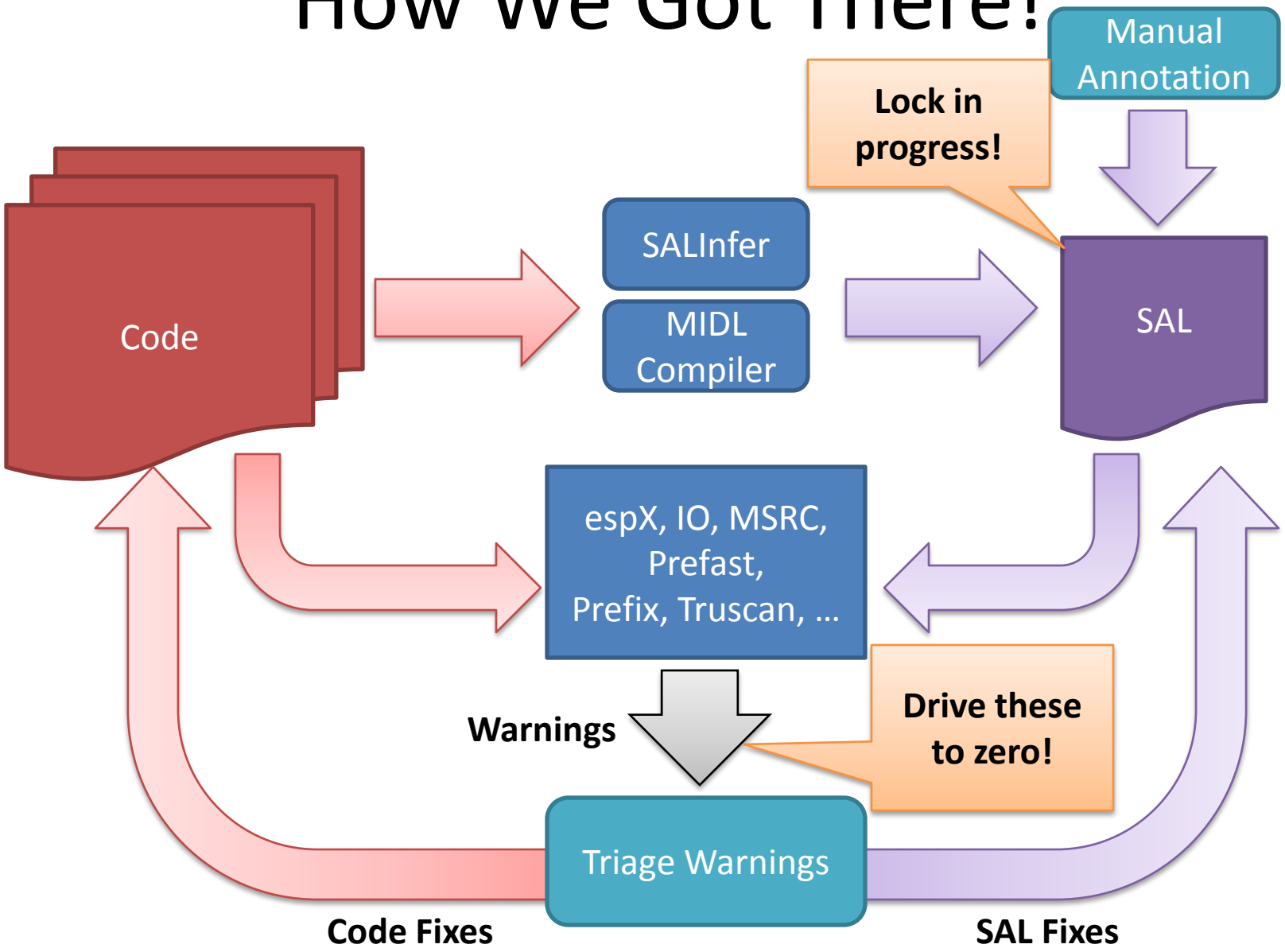
Headers for toy application only expose 1/5th of all Win32 APIs

Developers did more than the minimum required for security!

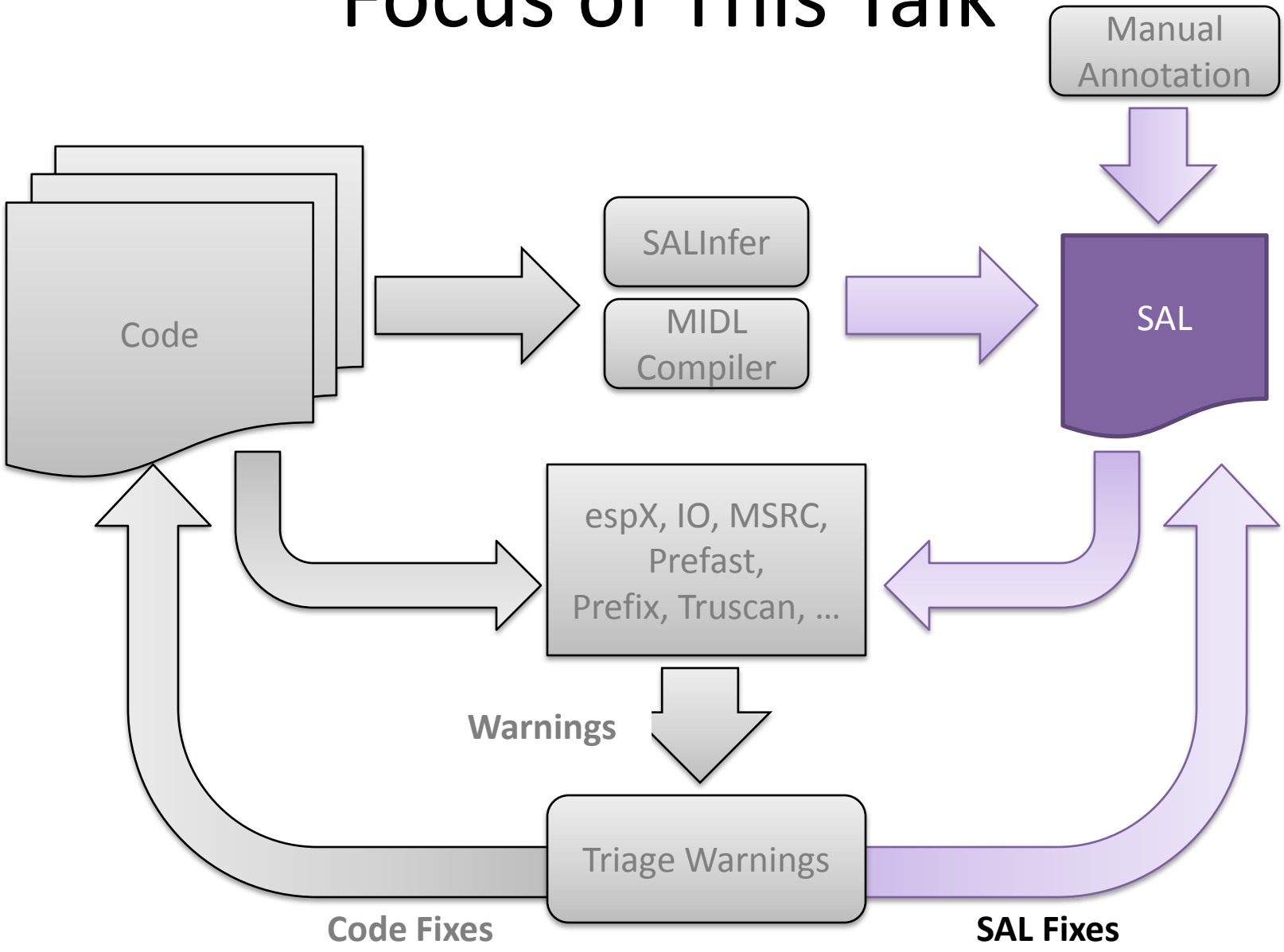
```
#include<tchar.h>
#include<windows.h>
#include<wincrypt.h>
#include<wininet.h>
#include<shlwapi.h>
#include<shlobj.h>

int _tmain(…)
{
 return 0;
}
```

How We Got There!



Focus of This Talk



Technical Design Goals

- Improves coverage and accuracy of static tools
- Locks in progress for the future
- Usable by an average windows developer
- Cannot break existing Win32 public APIs or force changes in data-structures (i.e. no fat pointers)

Technical Design Non-Goals

- No need to guarantee safety
- No need to be efficiently checked as part of normal foreground “edit-debug-compile” loop
- No need to handle all the corner cases
- No need to be “pretty”

Take a Peek Yourself!

For MSDN documented Win32 APIs start here

http://msdn2.microsoft.com/en-us/library/aa139672.aspx

or ~~Google~~ “Live Search” for them

Annotated headers can be download from

Vista SDK

first search hit for “Vista SDK”

MSDN Documentation for an API

Run-Time Library Reference

memcpy, wmemcpy

Copies bytes between buffers. These functions are deprecated because more secure versions are available; see memcpy_s, wmemcpy_s [http://msdn2.microsoft.com/en-us/library/wes2t00f(VS.80).aspx] .

```
void *memcpy(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemcpy(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest
New buffer.

src
Buffer to copy from.

count
Number of characters to copy.

Return Value

The value of **dest**.

Remarks

memcpy copies **count** bytes from **src** to **dest**; **wmemcpy** copies **count** wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy** is undefined. Use **memmove** to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns [http://msdn2.microsoft.com/en-us/library/ms717795(VS.80).aspx] .

The **memcpy** and **wmemcpy** functions will only be deprecated if the constant **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion statement in order for the functions to be deprecated, such as in the example below:

memcpy, wmemcpy, (cont)

Parameters

dest

New buffer.

src

Buffer to copy from.

count

Number of characters to copy.

Return Value

The value of **dest**.

Remarks

memcpy copies **count** bytes from **src** to **dest**; **wmemcpy** copies **count** wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy** is undefined. Use **memmove** to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or la er than the source buffer. For more information, see Avoiding Buffer Overruns [http://msdn2.microsoft.com/en library/ms717795(VS.80).aspx] .

The **memcpy** and **wmemcpy** functions will only be deprecated if the const **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion sta in order for the functions to be deprecated, such as in the example below:

For every API there's usually a wide version. Many errors are confusing "byte" versus "element" counts

Just say "No" to bad APIs.

Not all the information is relevant to buffer overruns.

msdn

International Features

MultiByteToWideChar

Maps a character string to a wide character (Unicode UTF-16) string. The character string mapped by this function is not necessarily from a multibyte character set.

```
int MultiByteToWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
  int cbMultiByte,
  LPWSTR lpWideCharStr,
  int cchWideChar
);
```

This unfortunately is a typical Win32 API

```
int MultiByteToWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
  int cbMultiByte,
  LPWSTR lpWideCharStr,
  int cchWideChar
);
```

lpMultiByteStr

[in] Pointer to the character string to convert.

cbMultiByte

[in] Size, in bytes, of the string indicated by the *lpMultiByteStr* parameter. Alternatively, this parameter can be set to -1 if the string is null-terminated. Note that, if *cbMultiByte* is 0, the function fails.

If this parameter is -1, the function processes the entire input string, including the null terminator. Therefore, the resulting wide character string has a null terminator, and the length returned by the function includes the terminating null character.

If this parameter is set to a positive integer, the functi processes exactly the specified number of bytes. If the provided size does not include a null terminat a resulting wide character string is not null-terminated, and the returned length does not include the t ting null character.

Not so common pattern

lpWideCharStr

[out] Pointer to a buffer that receives the converted string

cchWideChar

[in] Size, in WCHAR values, of the buffer indicated by *lpWideCharStr*. If this value is 0, the function returns the required buffer size, in WCHAR values, including any terminating null character, and makes no use of the *lpWideCharStr* buffer.

Return Values

Returns the number of WCHAR values written to the buffer indicated by *lpWideCharStr* if successful. If the function succeeds and *cchWideChar* is 0, the return value is the required size for the buffer indicated by *lpWideCharStr*.

A common pattern

How to Solve a Problem like MultiByteToWideChar?

- Start with an approximate specification
- See how much noise and real bugs you find
- Power up the tools and refine until you find the next thing to worry about
- Need conditional null termination to handle case when cbMultiByte is -1
- Buffer size weakening handles cbMultiByte 0 case

BCryptResolveProviders

The **BCryptResolveProviders** function obtains a collection of all of the providers that meet the specified criteria.

```
NTSTATUS WINAPI BCryptResolveProviders(
  LPCWSTR pszContext,
  ULONG dwInterface,
  LPCWSTR pszFunction,
  LPCWSTR pszProvider,
  ULONG dwMode,
  ULONG dwFlags,
  ULONG* pcbBuffer,
  PCRYPT_PROVIDER_REFS* ppBuffer
);
```

```
NTSTATUS WINAPI BCryptResolveProviders(
  LPCWSTR pszContext,
  ULONG dwInterface,
  LPCWSTR pszFunction,
  LPCWSTR pszProvider,
  ULONG dwMode,
  ULONG dwFlags,
  ULONG* pcbBuffer,
  PCRYPT_PROVIDER_REFS* ppBuffer
);
```

pcbBuffer

[in, out] A pointer to a **ULONG** value that, on entry, contai to by the *ppBuffer* parameter. On exit, this value receives either the required size, in bytes, of the buffer.

Optional reference argument!

ppBuffer

[in, out] The address of a **CRYPT_PROVIDER_REFS** [http://msdn2.microsoft.com/en-us/library/aa376232.aspx] pointer that receives the collection of providers that meet the specified criteria. If this parameter is NULL, this function will return STATUS_BUFFER_TOO_SMALL and place in the value pointed to by the *pcbBuffer* parameter, the required size, in bytes, of all the data.

If this parameter is the address of a NULL pointer, this function will allocate the required memory, fill the memory with the information about the providers, and place the pointer to this memory in this parameter. When you have finished using this memory, free it by passing this pointer to the **BCryptFreeBuffer** [http://msdn2.microsoft.com/en-us/library/aa375445.aspx] function.

If this parameter is the address of a non-NULL pointer, this function will copy the provide buffer. The *pcbBuffer* parameter must contain the size, in bytes, of the entire buffer. If t enough to hold all of the provider information, this function will return STATUS_BUFFER_TOO_SMALL.

Optional buffer!

Remarks

A common pattern to communicate buffer sizes to callee

BCryptResolveProviders can be called e e callers must be executing at PASSIVE_LEVEL *IRQL* [http://msdn2.microsoft.com/en-us/library/ms721588.aspx] .

GetEnvironmentStrings

Retrieves the environment variables for the current process.

```
LPTCH WINAPI GetEnvironmentStrings(void);
```

Parameters

This function has no parameters.

Return Value

If the function succeeds, the return value is a pointer to the environment block of the current process.

If the function fails, the return value is NULL.

Remarks

The **GetEnvironmentStrings** function returns a pointer to a block of memory that contains the environment variables of the calling process. Each environment block contains the environment variables in the following format:

Var1=*Value1*\0
Var2=*Value2*\0
Var3=*Value3*\0
...
VarN=*ValueN*\0\0

Double null termination!

Does Your Head Hurt Yet?

Does Your Head Hurt Yet?

If only C had exceptions, garbage collection, and a better string type the Win32 APIs would be much simpler!

Does Your Head Hurt Yet?

I WISH IT DID!

The Next Best Thing

Use the .NET Win32 bindings until it does!

The Next Best Thing

So when are they going to rewrite Vista in C#?

So That's Why It Took Five Years!

Read up about the "Longhorn Reset"
http://en.wikipedia.org/wiki/Development_of_Windows_Vista

So That's Why It Took Five Years!

Intel and AMD will solve this problem eventually! Until then we have ***SAL***.

MultiByteToWideChar

```
WINBASEAPI
int
WINAPI
MultiByteToWideChar(
    __in UINT     CodePage,
    __in DWORD    dwFlags,
    __in_bcount(cbMultiByte) LPCSTR   lpMultiByteStr,
    __in int      cbMultiByte,
    __out_ecount_opt(cchWideChar) LPWSTR  lpWideCharStr,
    __in int      cchWideChar);
```

BCryptResolveProvider

```
NTSTATUS WINAPI
BCryptResolveProviders(
    __in_opt LPCWSTR pszContext,
    __in_opt ULONG dwInterface,
    __in_opt LPCWSTR pszFunction,
    __in_opt LPCWSTR pszProvider,
    __in ULONG dwMode,
    __in ULONG dwFlags,
    __inout ULONG* pcbBuffer,
    __deref_opt_inout_bcount_part_opt(*pcbBuffer, *pcbBuffer)
      PCRYPT_PROVIDER_REFS *ppBuffer);
```

GetEnvironmentStrings

```
WINBASEAPI
__out
__nullnullterminated
LPCH
WINAPI
GetEnvironmentStrings(
    VOID
    );
```

End of Section A

Questions?

From Types to Program Logics a Recipe for SAL

A story inspired by true events

A Recipe for SAL

1) Start with a simple Cyclone like type system
2) Slowly shape it into a powerful program logic for describing common Win32 APIs
3) Add some syntactic sugar and abstraction facilities
4) Mix in a lot of developer feedback
5) Bake it until it's properly done!

It's getting there but still needs some cooking!

Types vs Program Logic

- Types are used to describe the representation of a value in a given program state
- Program Logic describe transitions between program states

Aside: Each execution step in a type-safe imperative languages preserves types so types by themselves are sufficient to establish a wide class of properties without the need for program logic

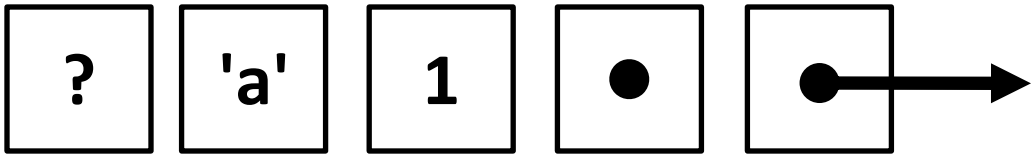
Concrete Values

Scalars

'\0',..,'a','b','c', …

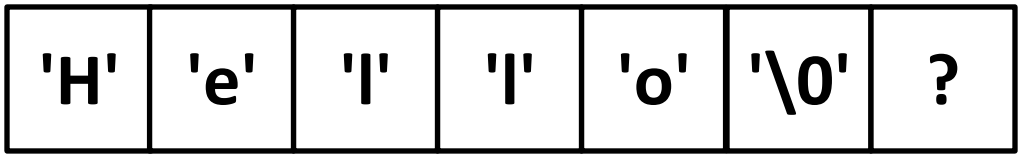
…,-2,-1,0, 1, 2, …

Cells



Pointers

Extent

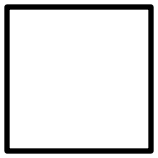


Abstract Values

Some Scalar

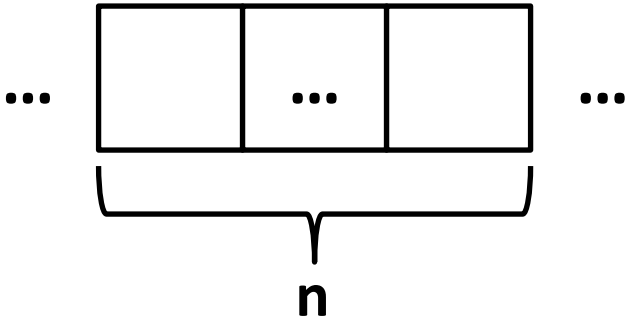
A,B, … ,X,Y,Z

Some Cell

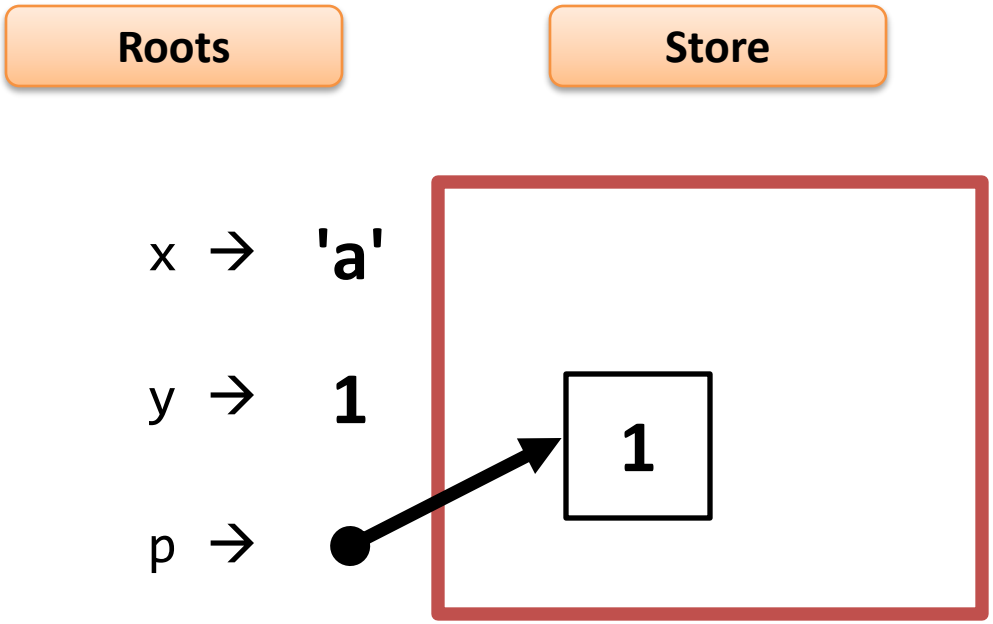


Some Pointer

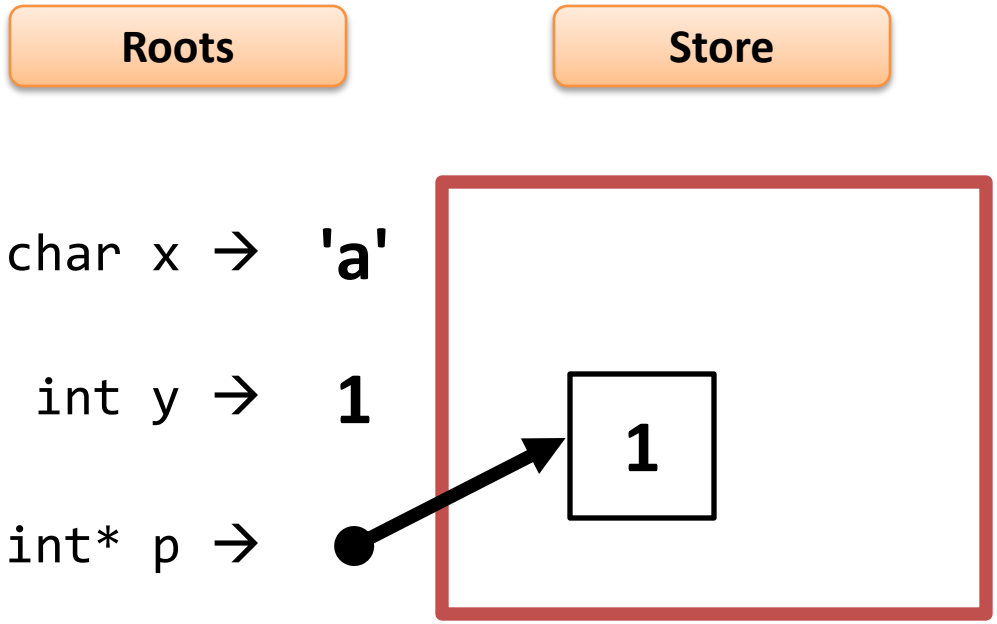
Some Extent



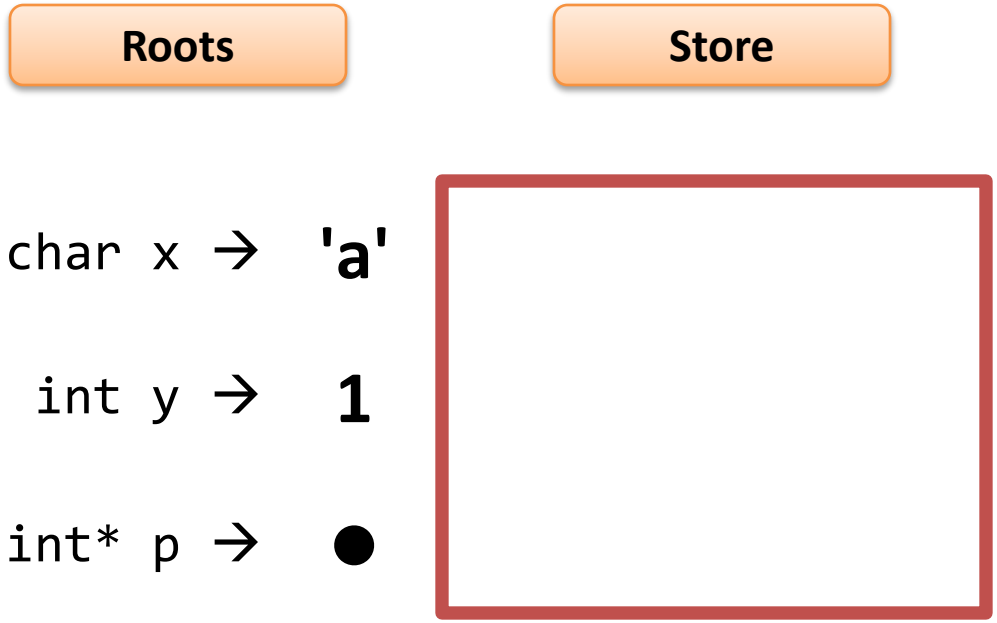
Program State



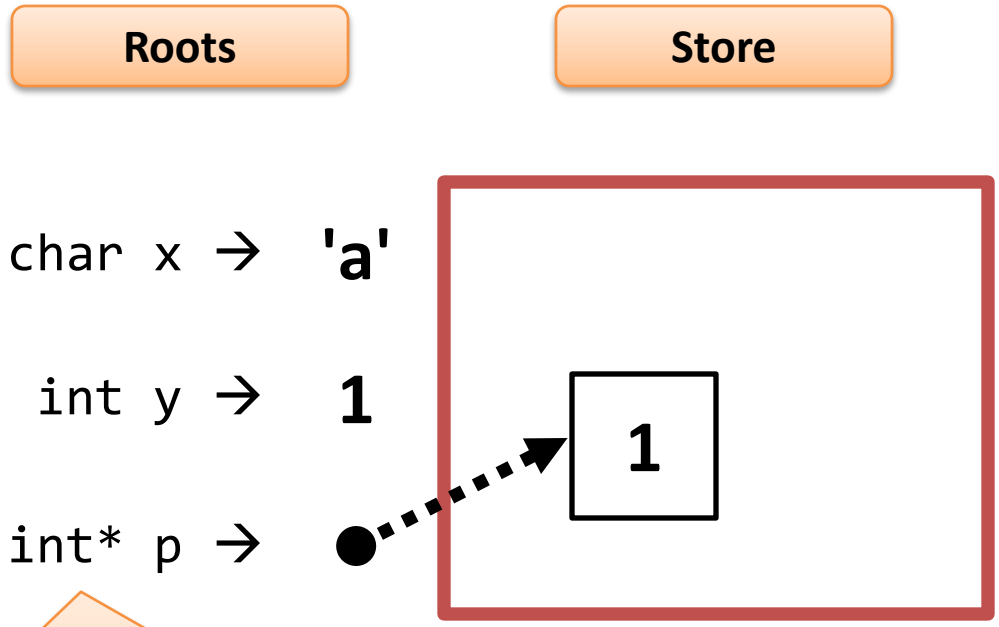
Well-Typed Program State



Well-Typed Program State

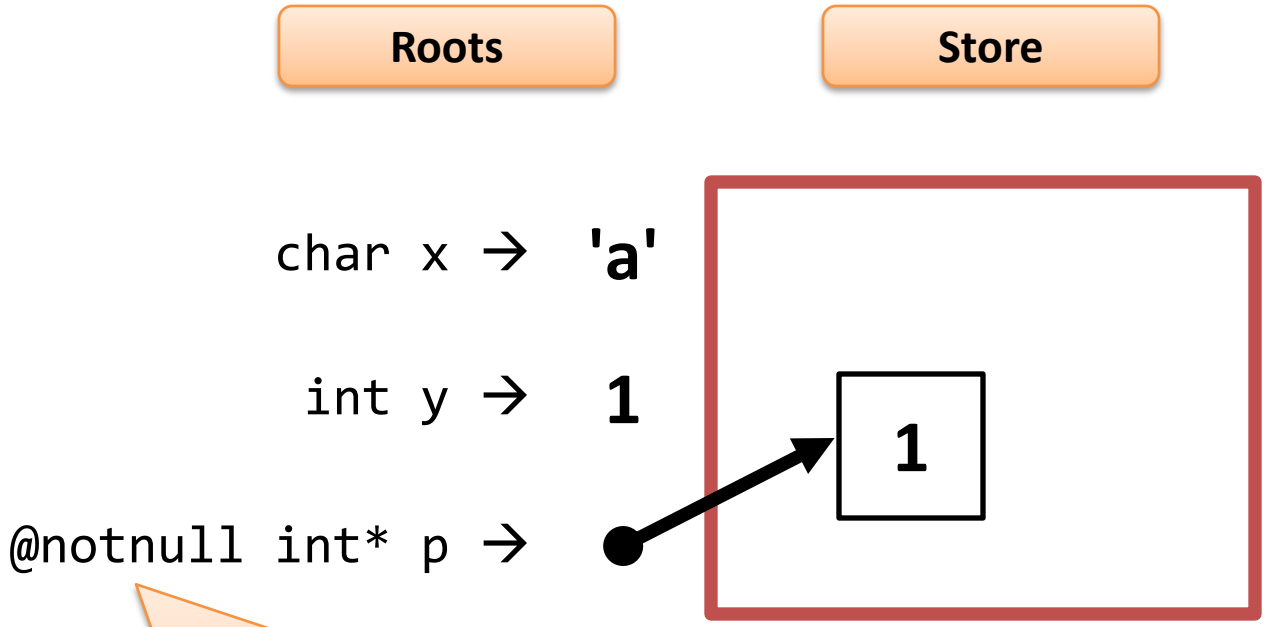


Well-Typed Program *States*



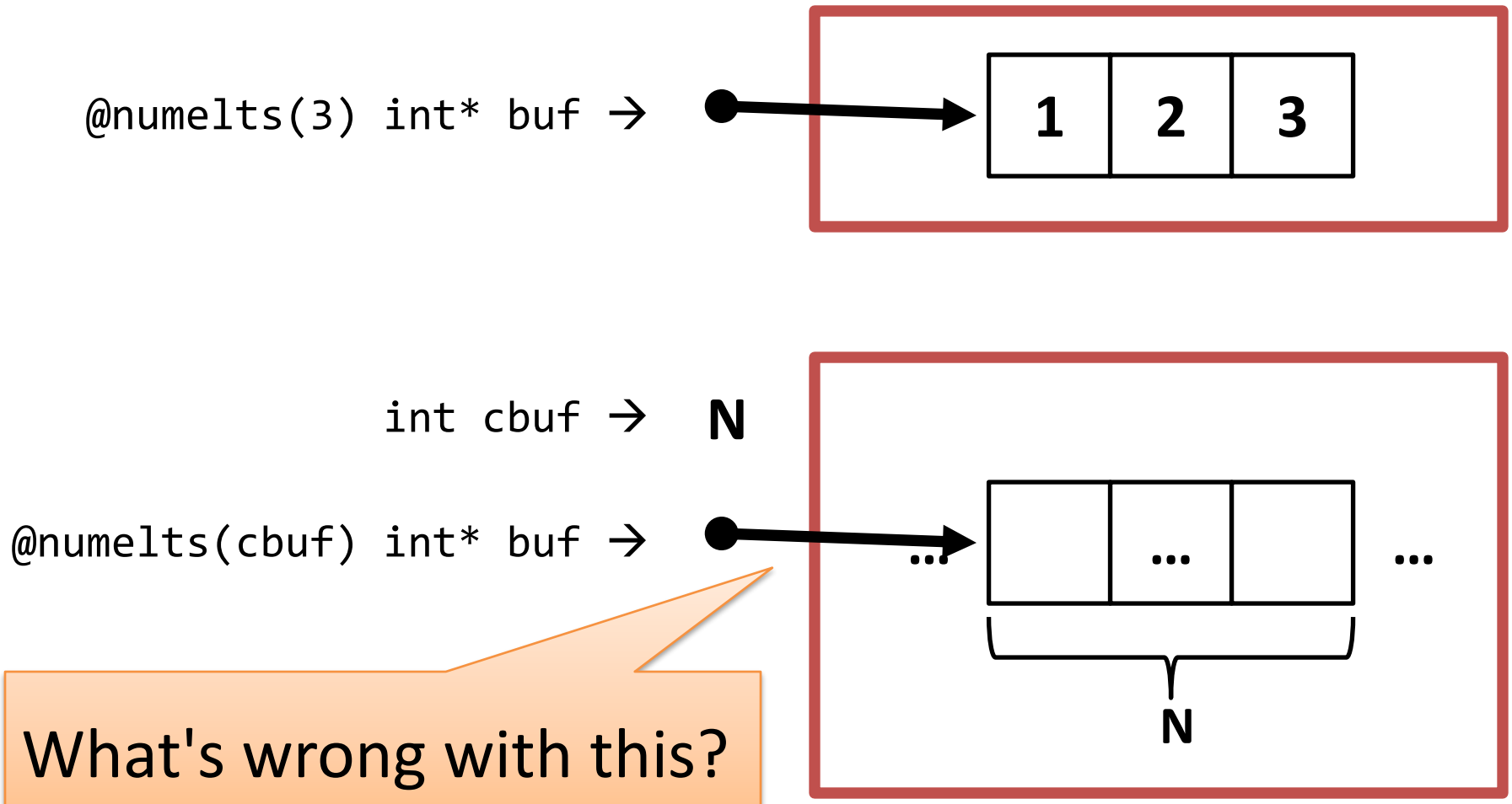
C types not descriptive enough to avoid errors

Well-Typed Program *States*

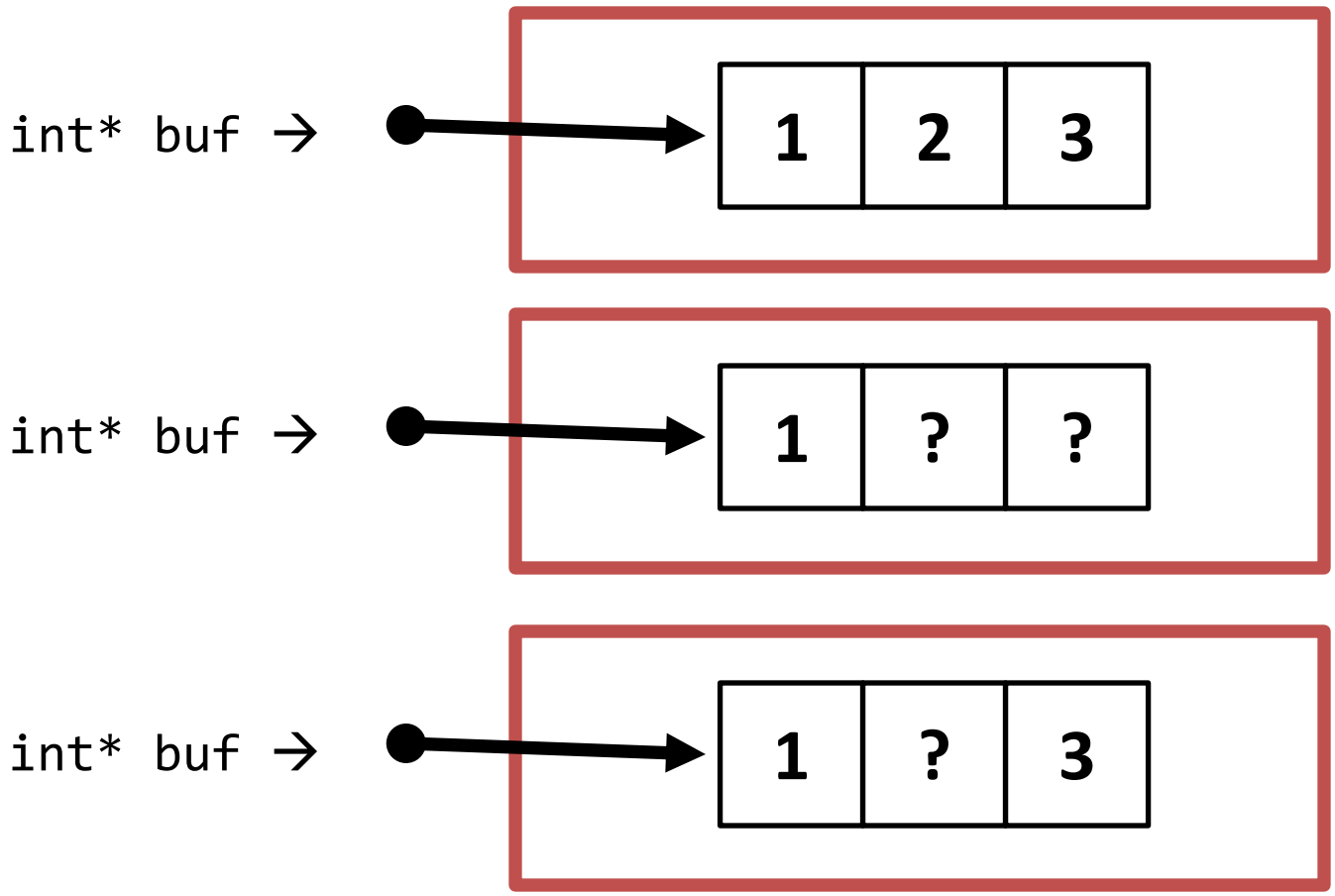


Use Cyclone style qualifiers to be more precise!

Generalizing @numelts

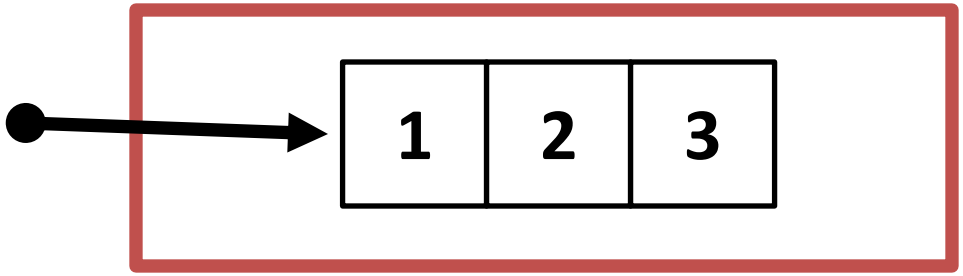


Is it Initialized or Not?

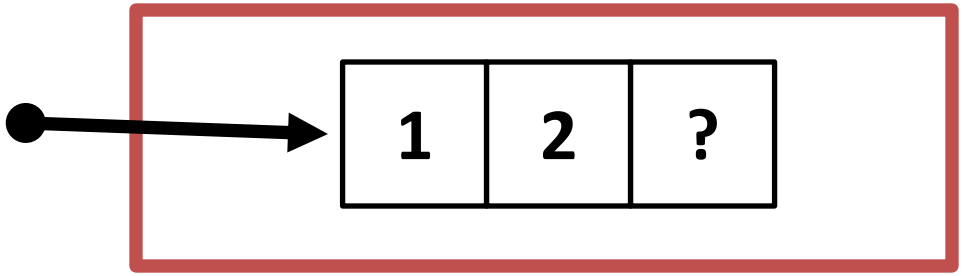


Define @numelts(e) as @extent(e,e)

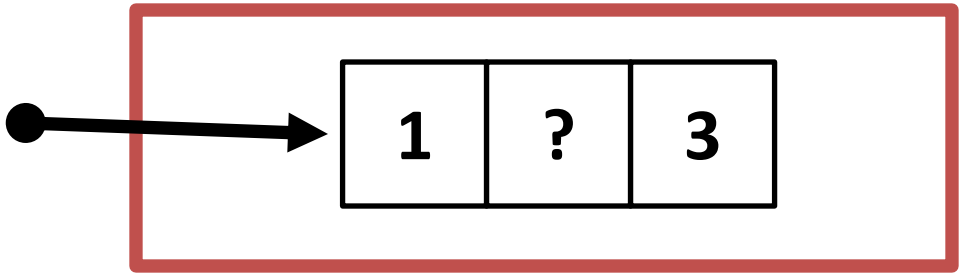
`@extent(3,3) int* buf →`



`@extent(2,3) int* buf →`

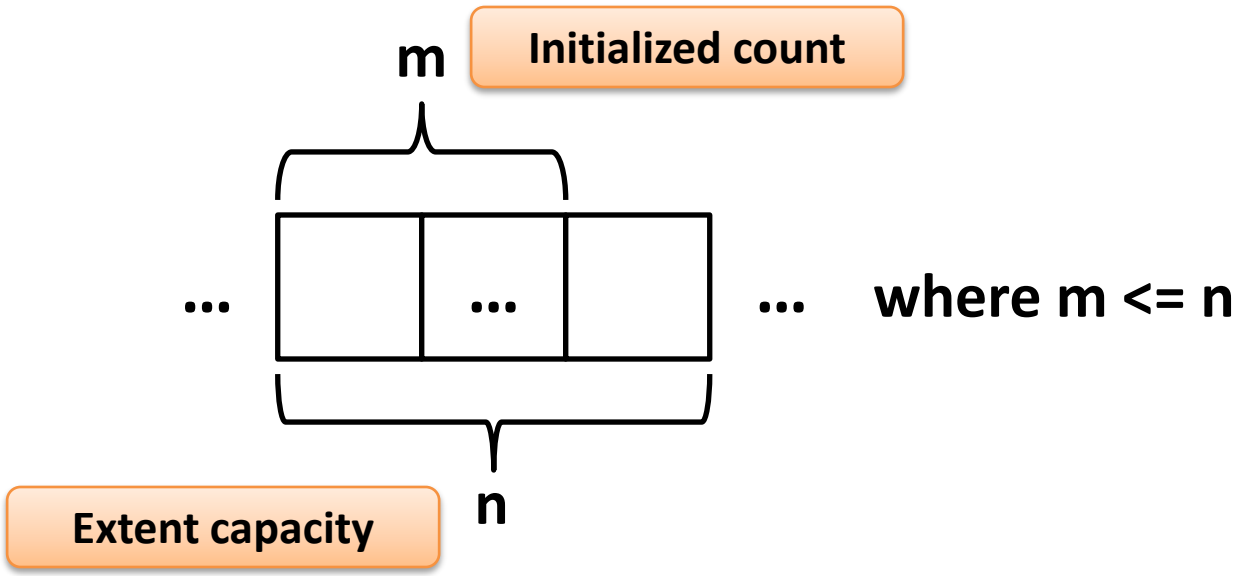


`@extent(??,3) int* buf →`

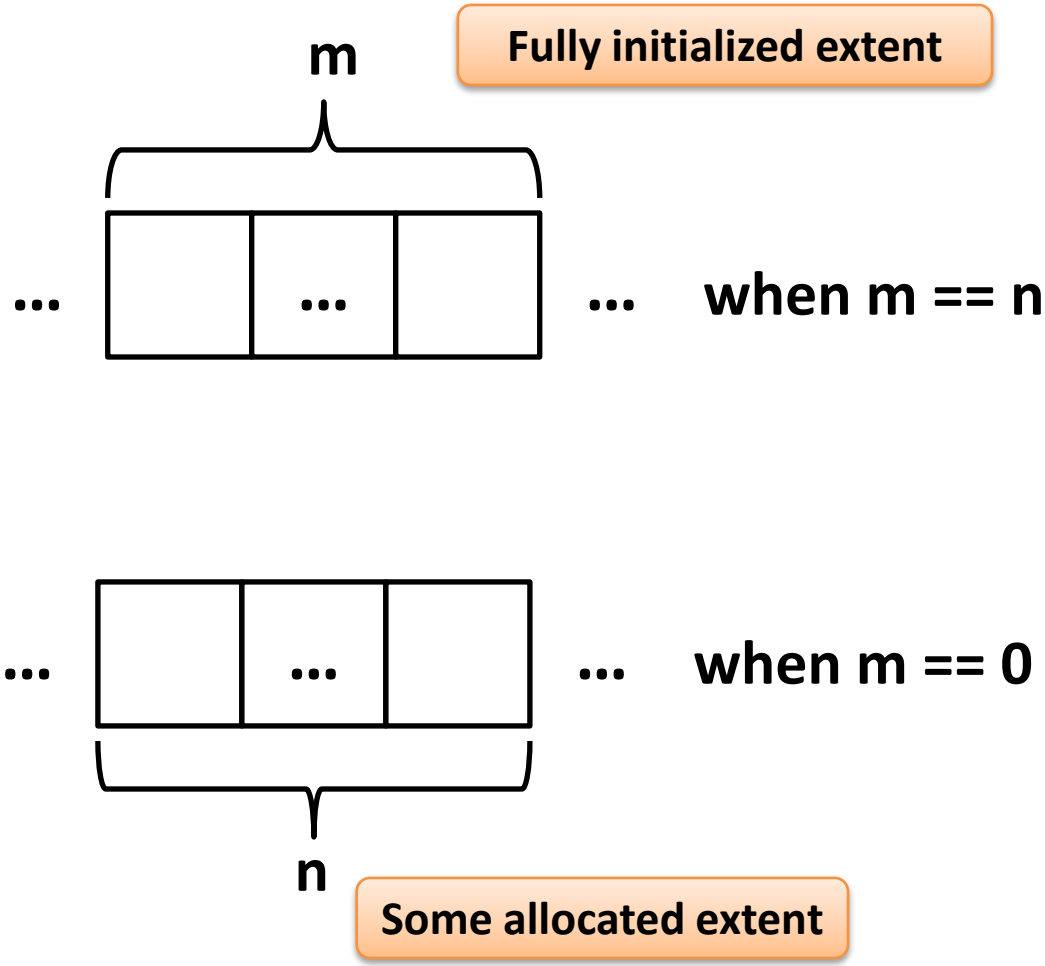


Just give up here!

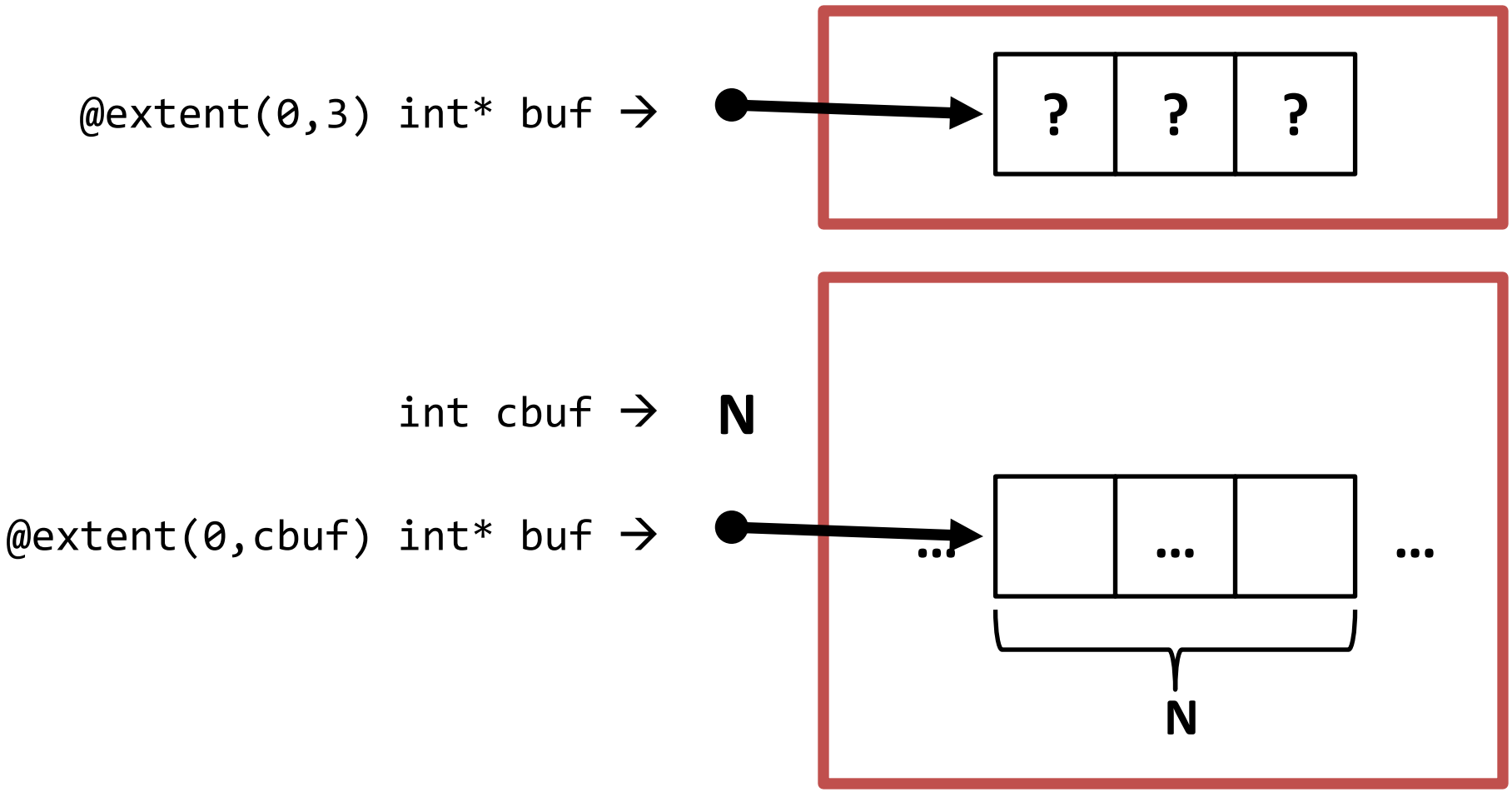
Refined Abstract Extent



Some Special Cases



@extent(count,capcity)



Qualified Types Useful for Win32 APIs

t ::= int | void | char | t* | q_1 … q_n t

q ::= @range(e_1, e_1) | @relop(e,op)

| @notnull | @nullable | @null | @readonly

| @numelts(e) | @alloced(e) | @extent(e_1, e_2)

| @bsize(e) | @balloced(e) | @bextent(e_1, e_2)

| @zeroterm | @zerozeroterm

op ::= == | <= | >= | !=

e ::= ….

A Qualifed Type for memcpy

```
@notnull @numelts(count)
void* memcpy(
  @notnull @alloced(count)
  void *dest,
  @readonly @notnull @numelts(count)
  const void *src,
  size_t count)
```

It seems to work? What's wrong?

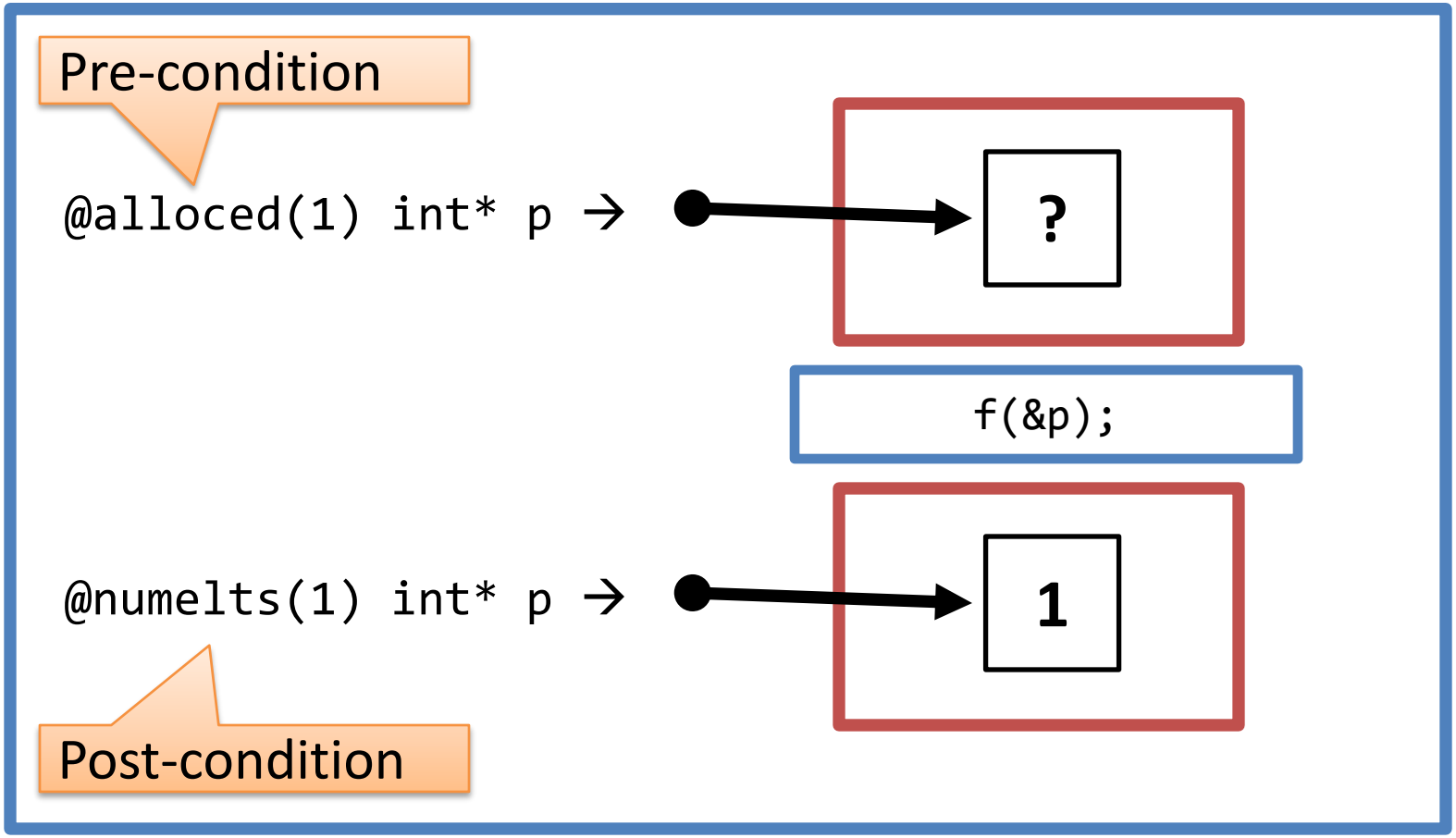
Which One is Right?

```
void f(@notnull @alloced(1) int *p) {
    *p = 1;
}

void f(@notnull @numelts(1) int *p) {
    *p = 1;
}
```

Types don’t capture the state transition!

Program State Transitions



Pre-post pair make a up a contract!

Contracts with Program Logics

```
void f( @Pre{ @notnull @alloced(1) }
        @Post{ @notnull @numelts(1) }
        int *p) {
        *p = 1;
}
```

Contracts with Program Logics

```
void f( @Pre{ @notnull @alloced(1) }
        @Post{ @numelts(1) }
        int *p) {
        *p = 1;
}
```

Simplify because C is call by value!

Contracts with Program Logics

```
void f( @Pre{ @notnull @alloced(1) }
        @Post{ @numelts(1) }
        int *p) {
        *p = 1;
}
```

Who in their right mind is going to write that!

Contracts with Program Logics

```
#define __out \
 @Pre{ @notnull @alloced(1) } \
 @Post{ @numelts(1) }

void f(__out int *p) {
        *p = 1;
}
```

C Preprocessor macros to the rescue!
Defined to empty string for compatibility.

Single Element Contracts

```
#define __in \
 @Pre{ @readonly @notnull @numelts(1) }

#define __out \
 @Pre{ @notnull @alloced(1) } \
 @Post{ @numelts(1) }

#define __inout \
 @Pre{ @notnull @numelts(1) } \
 @Post{ @numelts(1) }
```

Single Element Contracts

```
#define __in_opt \
 @Pre{ @readonly @nullable @numelts(1) }

#define __out_opt \
 @Pre{ @nullable @alloced(1) } \
 @Post{ @numelts(1) }

#define __inout_opt \
 @Pre{ @nullable @numelts(1) } \
 @Post{ @numelts(1) }
```

Contracts for Element Extents

```
#define __in_ecount(e) \
 @Pre{ @readonly @notnull @numelts(e) }

#define __out_ecount_part(cap,count) \
 @Pre{ @notnull @alloced(cap) } \
 @Post{ @extent(count,cap) }

#define __inout_ecount_part(cap,count) \
 @Pre{ @notnull @extent(count,cap) } \
 @Post{ @extent(count,cap) }
```

Note order of args

Note order of args

Contracts for Element Extents

```
#define __out_ecount_full(e) \
  __out_ecount_part(e,e)
#define __inout_ecount_full(e) …
/* opt versions */
#define __in_ecount_opt(e) …
#define __out_ecount_part_opt(cap,count) …
#define __inout_ecount_part_opt(cap,count) …
#define __out_ecount_full_opt(e) …
#define __inout_ecount_full_opt(e) …
```

Contracts for Byte Extents

```
#define __in_bcount(e) …
#define __out_bcount_part(cap,count) …
#define __inout_bcount_part(cap,count) …
#define __out_bcount_full(e) …
#define __inout_bcount_full(e) …
#define __in_bcount_opt(e) …
#define __out_bcount_part_opt(cap,count) …
#define __inout_bcount_part_opt(cap,count) …
#define __out_bcount_full_opt(e) …
#define __inout_bcount_full_opt(e) …
```

Developers can learn a small set of macros and be productive quickly

annotation	% total	% cum
__in	47.45%	47.45%
__out	10.37%	57.82%
__in_opt	6.48%	64.30%
__inout	5.42%	69.73%
__RPC__in	2.70%	72.42%
__out_ecount	2.57%	74.99%
__in_ecount	2.55%	77.54%
__RPC__out	2.45%	79.99%
__deref_out	2.17%	82.16%
__RPC__deref_out_opt	1.96%	84.12%
__out_opt	1.66%	85.78%
__in_bcount	1.17%	86.96%
__override	0.85%	87.81%
__RPC__in_opt	0.83%	88.63%
__out_bcount	0.72%	89.35%
__checkReturn	0.64%	89.99%
__inout_opt	0.59%	90.58%
__out_ecount_opt	0.56%	91.15%
__RPC__deref_out	0.56%	91.71%
__inout_ecount	0.51%	92.21%
__nullterminated	0.41%	92.62%
__in_ecount_opt	0.37%	92.99%
__deref_out_ecount	0.30%	93.29%
__RPC__in_ecount_full	0.30%	93.59%
__in_z	0.27%	93.87%
__out_bcount_opt	0.26%	94.12%
__deref_out_opt	0.25%	94.37%
__RPC__out_ecount_full	0.23%	94.60%
__in_bcount_opt	0.21%	94.82%
__reserved	0.20%	95.01%

Distribution of macros used across Vista source base.

Contract for memcpy

```
__out_bcount_full(count)
void* memcpy(
  __out_bcount_full(count)
  void *dest,
  __in_bcount(count)
  const void *src,
  size_t count);
```

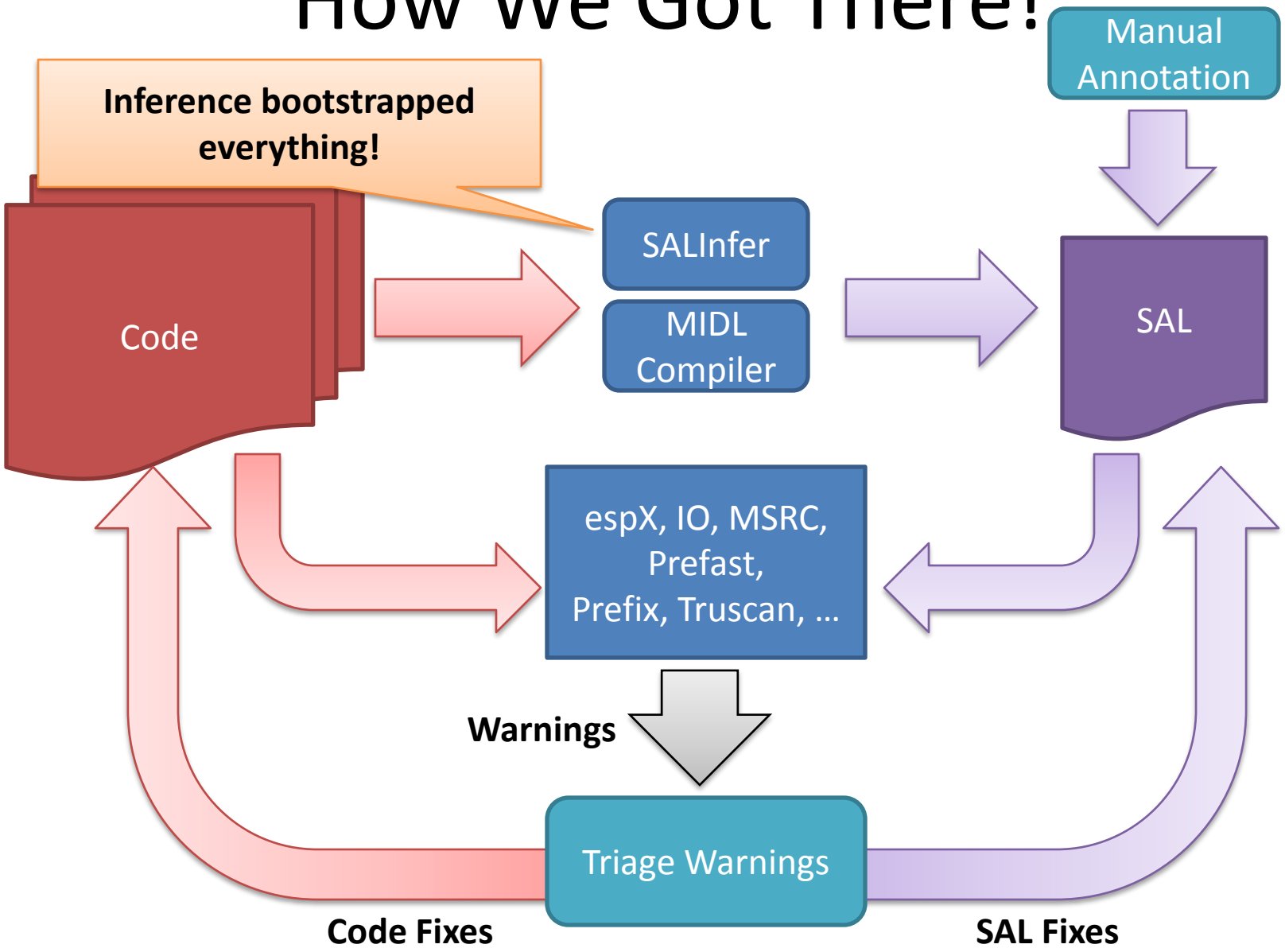
Ignore meaningless pre-condition

What about pointers to pointers?

```
void f( __out (@nullable int*)* p) {
    static int l = 3;
    if(…) *p = NULL;
    else *p = &l;
}
void f(__deref_out_opt int **p) { … }
```

Syntax makes applying ***automatically inferred*** annotations to legacy code tractable!

How We Got There!



What about Nested Pointers?

```
#define __deref_out_opt \
  @Pre{ @notnull @alloced(1) } \
  @Deref @Post { @nullable @numelt(1) }
```

Pushes context of assertion down a pointer level

Annotated Types for Win32 APIs

t ::= int | char | void | t* | t

at ::= a_1 … a_n t

p ::= @range(e_1, e_1) | … | @zerozeroterm

a ::= @Deref a| @Pre { p_1 … p_n } | @Post { p_1 … p_n } | p

op ::= …

e ::= ….

Annotated type split into annotations and type,
Not mixed in as type qualifiers

Actual primitive syntax is different. Just use the macros! Your code will be non-portable if you don't!

What About This Case?

```
bool f(__out_opt int *p) {
  if(p != NULL) {
    *p = 1;
     return true;
  }
  return false
}
```

Need to introduce conditional contracts!

Adding __success(cond)

- Most conditional behavior is related to error handling protocols (i.e. exceptions via return codes)
- Introduce specialized construct for this case

  __success(expr) f(…); means Post-conditions only hold when "expr" is true (non-zero) on return of function.
- Full conditional support on the roadmap!

Using Success

```
__success(return == true)
bool f(__out_opt int *p) {
  if(p != NULL) {
    *p = 1;
     return true;
  }
  return false
}
```

Is _opt the right thing?

Using Success Correctly!

```
__success(return == true)
bool f(__out int *p) {
  if(p != NULL) {
    *p = 1;
     return true;
  }
  return false
}
```

Annotate for successful case!

StringCchCat Function

StringCchCat is a replacement for strcat [http://msdn.microsoft.com/library/en-us/vclib/html/_crt_strcat.2c_.wcscat.2c_._mbscat.asp] . The size, in characters, of the destination buffer is provided to the function to ensure that **StringCchCat** does not write past the end of this buffer.

Syntax

```
HRESULT StringCchCat(
    LPTSTR pszDest,
    size_t cchDest,
    LPCTSTR pszSrc
);
```

Parameters

pszDest
: [in, out] Pointer to a buffer containing the string to which *pszSrc* is concatenated, and which contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest*.

cchDest
: [in] Size of the destination buffer, in *characters*. This value must equal the length of *pszSrc* plus the length of *pszDest* plus 1 to account for both strings and the terminating null character. The maximum number of characters allowed is STRSAFE_MAX_CCH.

pszSrc
: [in] Pointer to a buffer containing the source string that is concatenated to the end of *pszDest*. This source string must be null-terminated.

Contracts For StringCchCat

```
HRESULT StringCchCat(
  __post __nullterminated __out
  LPTSTR pszDect,
  __range(0,STRSAFE_MAX_CCH)
  size_t cchDest,
  __nullterminated __in
  LPCTSTR pszSrc);
```

Much more verbose than we'd like!

Types with Contracts For StringCchCat

```
typedef __nullterminated TCHAR* LPSTR;
typedef const LPSTR LPCSTR;
typedef __range(0,STRSAFE_MAX_CCH) size_t
  STRSIZE;

HRESULT StringCchCat(
  __out LPTSTR pszDect,
  __in STRSIZE cchDest,
  __in LPCTSTR pszSrc);
```

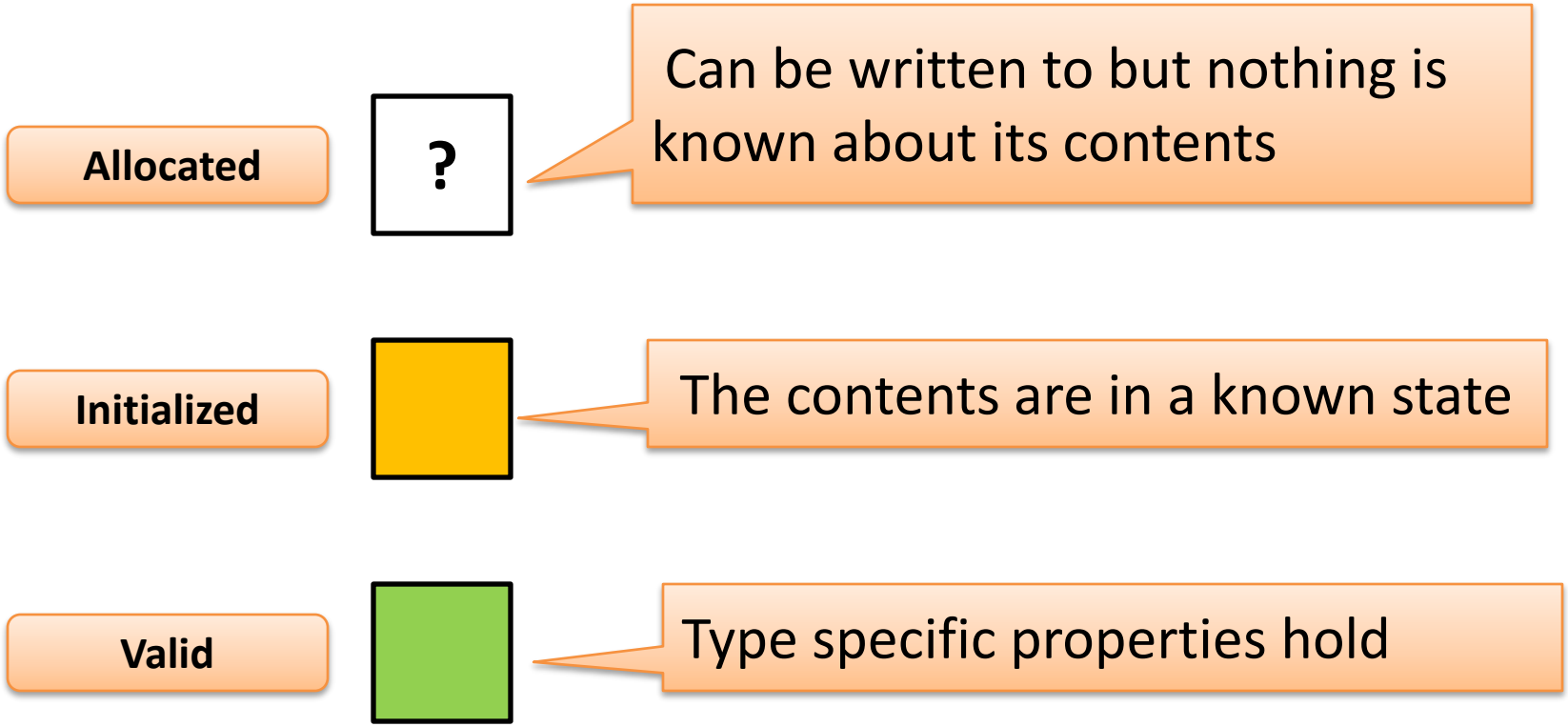
Must mean null terminated only in post condition!

New primitive @valid

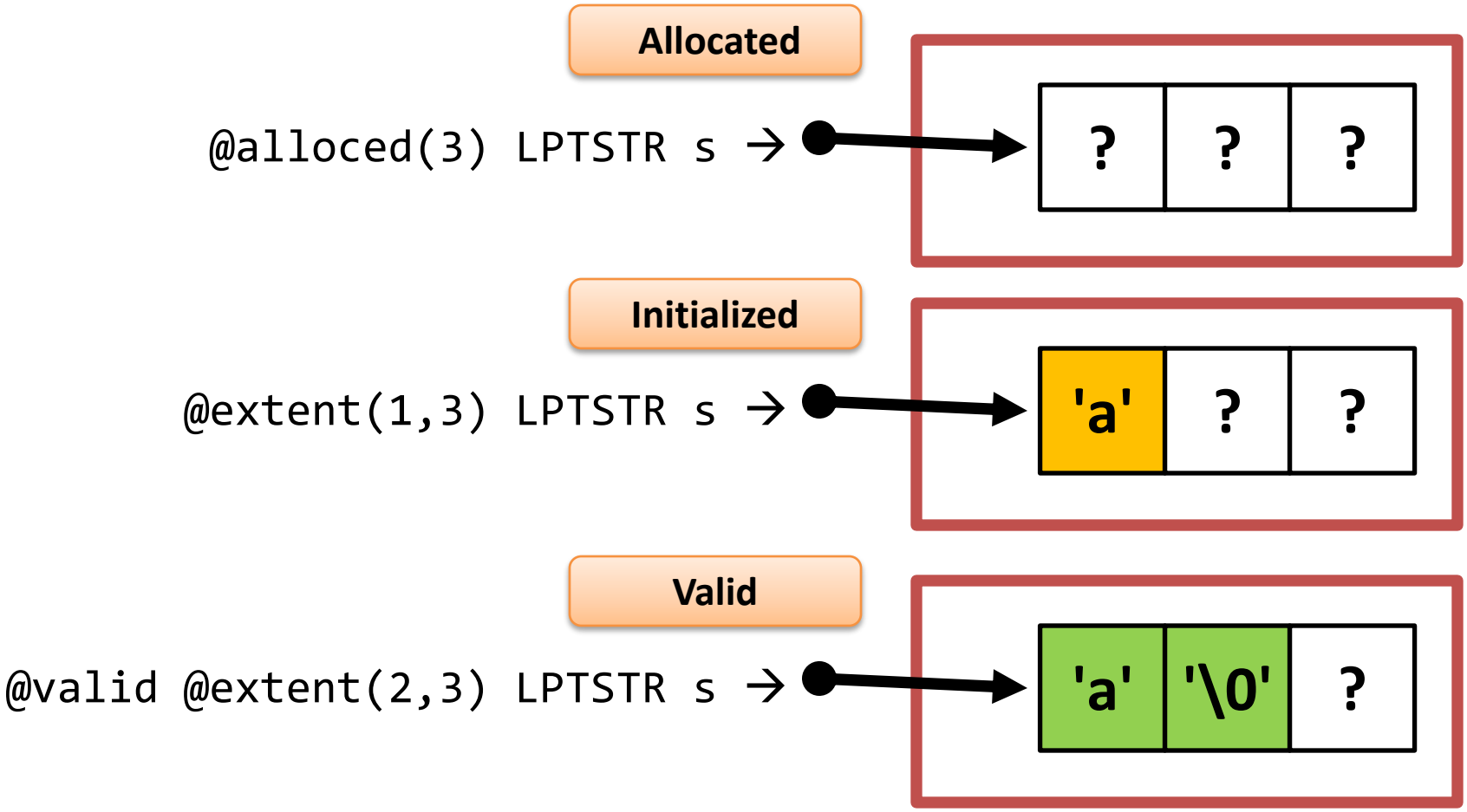
```
typedef @zeroterm TCHAR* LPSTR;
void f( @Pre{@notnull @alloced(1)}
        @Post{@valid @numelts(1)}
        LPSTR s) {
  s[0]='\0';
}
```

Annotations associated with types only happen when an extent is "valid"

Memory Semantics Revisited



Lifecycle of a LPTSTR



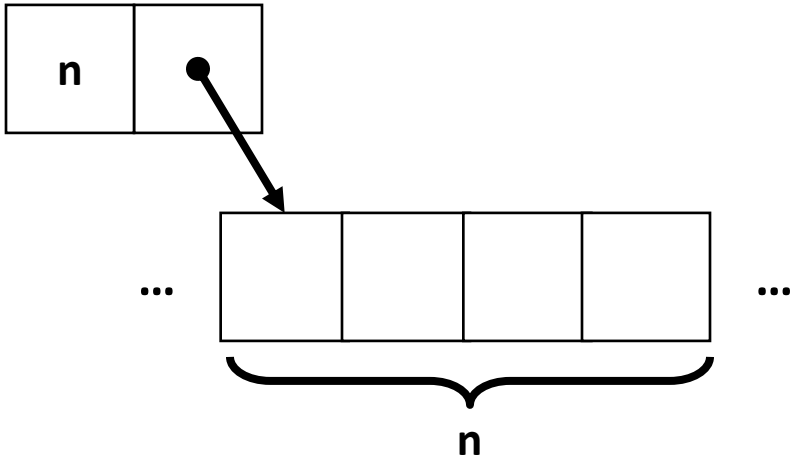
Validity: Related Work

- Validity is a lot like the Boogie methodology used in Spec#
  - Not as general since validity is just baked into macros
  - Many things are conditionally valid because of __success
  - Full conditional pre/post will allow more flexibility
- Even without it we can do some interesting with Objects
  - Treat them like structs!
  - Added in a few defaults

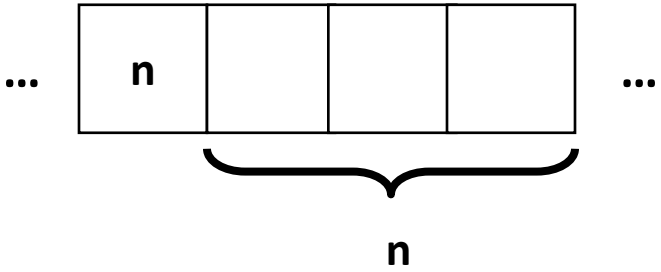
Structure Annotations

- Describes properties of buffers embedded in structs/classes
- Three scenarios supported
  - Outlined structure buffers
  - Structs with inline buffers
  - Header structs
- Structure descriptions interact with __in, __out, and __inout to determine pre/post rules for functions using structure buffers

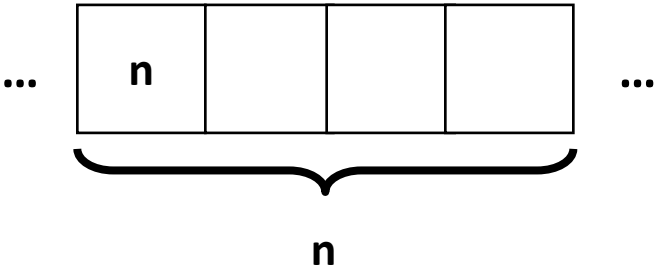
```
struct buf {
  int n;
__field_ecount(n)
  int *data;
};
```




```
struct ibuf {
  int n;
__field_ecount(n)
  int data[1];
};
```

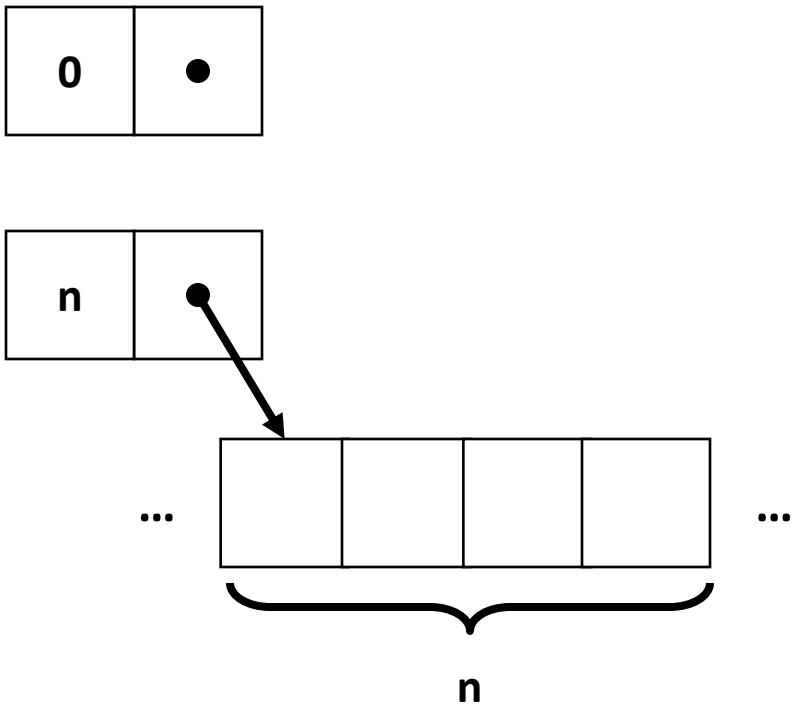



```
__struct_bcount(n * sizeof(int))
struct hbuf {
  int n;
  int data[1];
};
```



Zero Sized Buffers and NULL

```
struct buf {
  int n;
__field_ecount(n)
  int *data;
};
```



_opt versions available but generally not needed

SAL Annotations for Classes

```
class Stack {
public:
 Stack(int max);   // Stack(__out Stack *this,int  max);
 int Pop();        // int Pop(__inout Stack *this);
 void Push(int v); // void Push(__inout Stack *this,int v);
 ~Stack();         // treated specially
private:
 int m_max;
 int m_top;
 __field_ecount_part(m_max,m_top)
 int *m_buf;
};
```

Conclusions

- Developers will accept the use of appropriate light weight specifications!
- But must understand the problem and tailor custom solutions
- Generic recipe:
  1) Write the problem down.
  2) Think real hard.
  3) Write the solution down.
  4) Repeat!

Questions?

http://www.microsoft.com/cse